Using the Common Session Layer

Kabira Infrastructure Server 4.2.5

Trademarks

Feedback

This document is intended to address *your* needs. If you have any suggestions for how it might be improved, or if you have problems with this or any other Kabira documentation, please send e-mail to pubs@kabira.com. Thank you!

Feedback about this document should include the reference “Using the Common Session Layer” with the creation date.
Created on: February 2, 2005

Contents

Overview.. 1

Description of a serial interface.. 1

Description of the Common Session Layer.................................... 1

Integration with cartridge framework.. 2

Common Session Layer features... 2

Command.. 2

Serial Protocol specific features.. 2

General exchange overview.. 4

Step by step execution of a command.. 4

Extract the buffer to send... 4

Send the buffer.. 4

Receive the response.. 5

TRN / XML state machine overview... 6

TRN overview.. 7

XML state machine overview... 7

Using the TRN parser and complex guards................................ 9

Configuring the Cartridge... 11

Note on the serial protocol.. 11

Create one or more types using the factory.................................. 11

Configure the type... 12

Creating the instance.. 12

Configuring the instance... 12

Default mandatory parameters from cartridge......................... 13

Serial protocol component configuration................................ 13

Parameters added by CSL... 13

Configuration example... 14

Feature and parameter description................................ 15

Login and logout sequence... 15

Presentation.. 15

CSL sequence description... 15

Login sequence.. 16

Sleeping during login sequence... 16

Logout sequence.. 17

Note on more complex sequences.. 17

Closing session on max attempts... 17

Description... 17

Example... 17

Use case... 18

Configuring exchanges... 18

Append a string to sent data.. 19

Skipping the post command string... 19

Configuring early tokens.. 19

Pre-command setup 20
Handling early token absence 20

Implementing and using the cartridge 22
Defining work order flows 22
States 22
Transitions 23
CSL-enabled session layer input parameters 23
Defining configurable parameters 23
Creating the work order configuration file 24
Writing the TRN file 24

Troubleshooting / FAQ 25
How to debug login sequences ? 25
How to troubleshot slow exchanges ? 25
When and how does CSL publish logs ? 26
Publication time 26
Published logs 26
Sent/received buffer format 26
How to configure a login or logout sequence with a special character inside ? 26
How to develop serial protocol components ? 27
Does CSL handles timeout ? 27

External resources 28

References 29

Dependencies 30

Overview

This chapter provides an overview of the Common Session Layer (CSL). It will contain the following sections:

- Definition of a serial protocol component.
- Description of the CSL.
- Integration with cartridge framework.
- CSL features.

Description of a serial interface

A *serial interface* is an interface which transports “bag of bytes” or “MML commands” from a point to another. Common examples of serial interfaces are “telnet”, “tcp”, “pad” and “x25”: the data exchanges on these protocol are seen as sequence of bytes.
Pad with regards to x25 can be compared to telnet with regards to tcp.

These interfaces usually support 2 kinds of operations:
“send” to send data.
“recv” to receive data.

A *serial protocol component* is therefore a component which implement a “send/recv” interface. Such an interface is defined as a generic interface by the Common Session Layer.

Note that the default cartridge archive contains a *serial protocol component* for telnet protocol, and that is SProTelnet component.

Description of the Common Session Layer

Many of the protocols based on “telnet” or “pad” will share common constraints such as:

- Send a command on a connection and wait for a response on this connection.
- Another command cannot be sent on a connection while waiting for a response.

With these constraints, a lot of the logic and features that can be built on top of these serial protocols do not depend on the transport details: they are generic. As an example, retrying a “recv” until a particular keyword is received is not related to pad or telnet.

The Common Session Layer gathers these common requirements and, given a basic *serial protocol component* able of handling transport level aspects of the communication, allow users to quickly describe the protocol used by a network element by configuration.

Integration with cartridge framework

In the cartridge world, a session layer is a high level representation of a communication protocol. The Common Session Layer provides this high level representation to serial protocols.

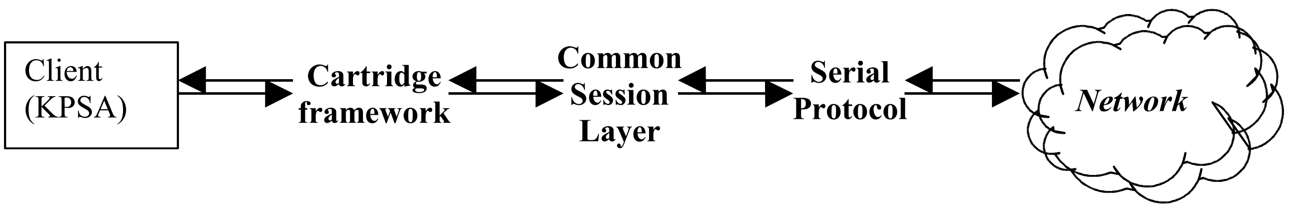


Figure 1: Integration

The different actors are:

- Client: this is a KPSA application which needs to send request to a Network Element using a *serial protocol*.
- Cartridge framework: this is the generic cartridge utility. (see also cartplugin idlos documentation).
- Common Session Layer: this is the component containing common logic to handle communications through *serial protocols*.
- Serial Protocol: this is the component containing specific *serial protocol* implementation (SProTelnet for telnet, SProPad for pad, etc).

All CSL-enabled session layers will have "CSL_" prefixed to their names. As an example, the Session Layer for telnet is "CSL_SProTelnet". Please refer to the user guide of the serial protocol you are using for more information.
Note that the "SPro" prefix stands for **S**erial **Pro**tocol.

Common Session Layer features

Command

CSL-enabled session layers will only support *one* type of command: sending a command buffer and receiving a response buffer. Therefore, the command name is not important here, only data sent and received are important.

Serial Protocol specific features

The following features related to serial protocol are supported:

- Login sequence.
- Logout sequence.
- Pre-command sequence.
- Early tokens detection.

Overview : Common Session Layer features

- Retries on error.
- Post command string appending.
- Sleep after send upon login.
- Automatic closing of session upon max attempts reached.

General exchange overview

This chapter will describe how is executed a command inside a Common Session Layer. Understanding this processing is key to effective configuration of the Common Session Layer.

This chapter will cover:
- Step by step execution of a command.
- TRN / XML state machine overview.

Step by step execution of a command

When a command is received by a CSL-enabled session layer, a send/receive mechanism is engaged:

Extract the buffer to send

The command to send is extracted from the work order NVSet parameter "#NE_COMMAND". The value list of an NVSet can contain multiple parameters, but only the first one (index zero) is used here. If a post command string is defined, it is appended to this buffer.

If a pre-command sequence is defined, it is executed. (see next chapters for more information)

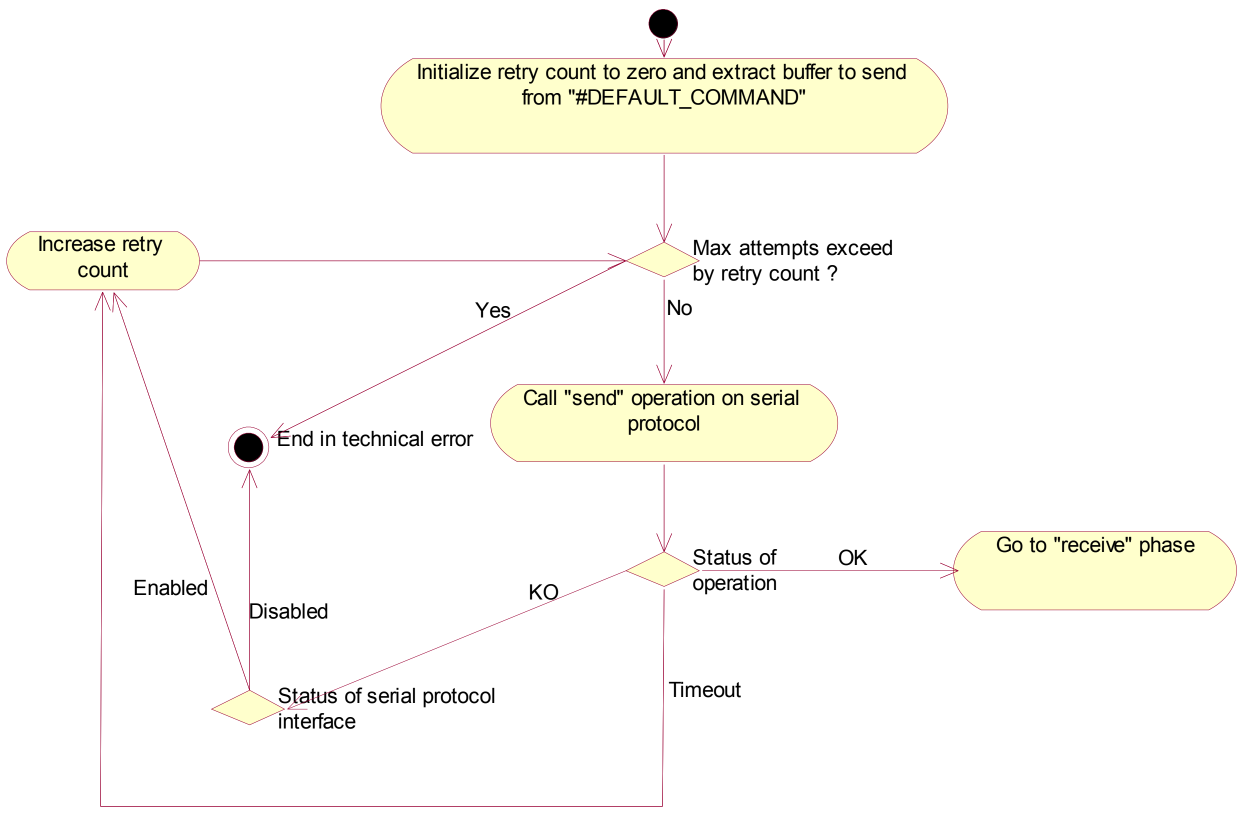
Send the buffer

The CSL tries to send the buffer to the network element using the *serial protocol component*. Three cases can occur here, based on the serial protocol component response:
1. The result is OK: the CSL will go to the receiving step.
2. The result is Timeout: the CSL will try again, up to the "MaxAttemptNumber" configuration parameter.
3. The result is KO: the CSL will query the status of the serial protocol interface.
   - If the interface is up and running, the CSL will try again, up to the "MaxAttemptNumber" configuration parameter.
   - If the interface is down, the session layer session is declared as disabled, and the command ends in a technical error.

(see following graph)

General exchange overview : Step by step execution of a command

The serial protocol interface can communicate its status “up and running” / “down” to the upper level, to the CSL layer.

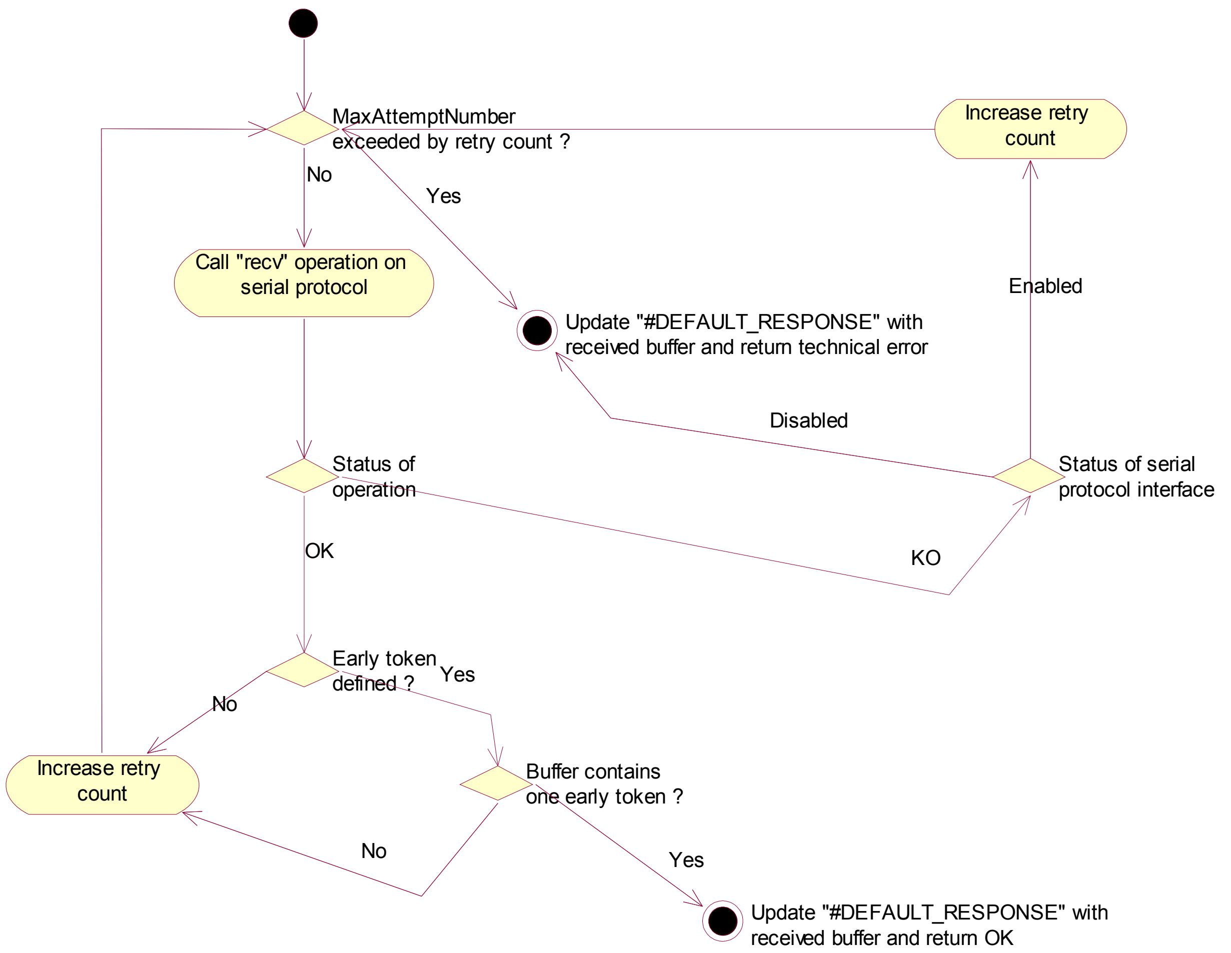

Initialize retry count to zero and extract buffer to send from "#DEFAULT_COMMAND"
Increase retry count
Max attempts exceed by retry count ?
Yes
No
Call "send" operation on serial protocol
End in technical error
Status of operation
OK
Go to "receive" phase
Enabled
Disabled
KO
Status of serial protocol interface
Timeout

Receive the response

The CSL will then try to receive response buffer from the network element using the *serial protocol component*. Again, 3 cases can occur here:

1. The result is OK: the CSL will consider it has a valid buffer. Based on configuration for “EarlyTokens”, the CSL might decide to receive again, or not. For example:
   - If no “EarlyTokens” are configured, the result is considered as KO, and the receive process will be retried up to “MaxNumberOfAttempts”. The result of the processCommand on the Session Layer will be technical KO.
   - If one of the configured “EarlyTokens” is present in the received buffer, the result of the processCommand on the Session layer is OK.
   - In any case the “#NE_RESPONSE” value of the NVSet is filled with the received buffer.
   - See chapter on “XML state machine” for more information.
2. The result is Timeout: the CSL will try again, up to the “MaxAttemptNumber” configuration parameter.
3. The result is KO: the CSL will query the status of the serial protocol interface.

- If the interface is up and running, the CSL will try again, up to the “MaxAttemptNumber” configuration parameter.
- If the interface is down, the session layer session is declared as disabled, and the command ends in a technical KO.


MaxAttemptNumber exceeded by retry count ?
Increase retry count
No
Yes
Call "recv" operation on serial protocol
Update "#DEFAULT_RESPONSE" with received buffer and return technical error
Enabled
Disabled
Status of operation
Status of serial protocol interface
OK
KO
Early token defined ?
Yes
No
Increase retry count
Buffer contains one early token ?
No
Yes
Update "#DEFAULT_RESPONSE" with received buffer and return OK

(see following graph)

Note that the received buffer grows with all “recv” operations during the receive phase. It is reset when the command completes.

As shown, there is only one buffer sent/received by command executed by a CSL-enabled session layer.

TRN / XML state machine overview

Using both TRN and XML state machines, very complex work orders (sequence of commands on a CSL-enabled session layers, i.e. sequences of buffer sent / response received) can be created.

TRN overview

CSL-enabled session layers will use “bag of bytes” sent over the network to communicate with a network element.
Very often, this “bag of bytes” will be composed of fixed parts and dynamic parts extracted from the work order NVSet.

Therefore, cartridge embeds a TRaNslation and parsing utility, called the Common Formatter (CMF).
This utility is driven by a configuration file, the TRN file.

A TRN file contains formatting and parsing directive, associated to either a work order or a state inside an XML state machine (see below).

For example, if the integration with a network element needs to issue a “ls” command on a specific directory given in the work order NVSet, the TRN will look like:

```
FORMAT_COMMAND:ls <directoryPath>
```

Where “directoryPath” is the name of the variable in the work order NVSet.
See cmf idlosdoc documentation for more information on the formatting and parsing ability.

XML state machine overview

A CSL-enabled session layer handles low level command / response interaction over a serial protocol. By integrating an XML state machine, more complex exchanges can be built, and presented to a client application as a single operation on a network element.

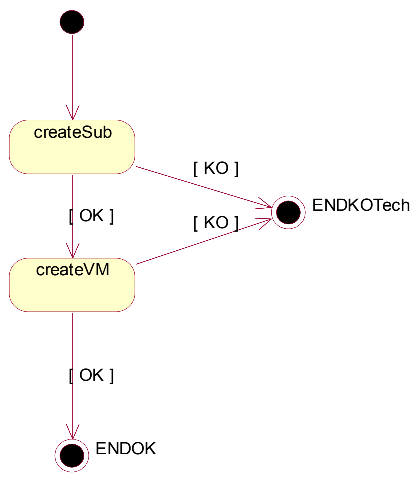
Consider the example of creating a new subscriber GSM line on an Intelligent Network element: 2 steps are required, which translates to 2 command / response interactions.
1. Create the subscriber account.
2. Create the voice mail account.

The parameters on the NVSet will be “SubscriberMSISDN” and “VoiceMailMSISDN”.
And the unit commands to create them are:

```
# create a subscriber line with MSISDN 0612341234
create subscriber msisdn=0612341234

# create associated voicemail, with MSISDN voicemail 0612349999
create voicemail msisdn=0612349999 subscriber=0612341234
```

This can be done using XML state machine configured in cartridge.
In Rational Rose, the work order flow will look like:

General exchange overview : TRN / XML state machine overview



This assumes that:

- Network element returns either OK for success and KO for failure.
- No rollback is needed. (in other words, if the creation of the voice mail fails, we do not rollback the creation of the subscriber).

Notes:

- The default guards in the work order flows consist in checking that the "#NE_RESPONSE" NVSet parameter contains the guard string. The integration with CSL-enabled session layer is therefore very easy as you can specify error / success conditions directly.
- More complex guards can be used on NVSet parameter names. This type of guards may be used when integrating with a parser such as the TRN parser. See next chapter for more details.

The corresponding TRN file will need to format the commands to send to the network element. We will call our work order "CreateFullSubscriber".

```
#
# Subscriber with voice mail creation work order:
#

# First step: subscriber creation:
STATE:CreateFullSubscriber,createSub
FORMAT_COMMAND:create subscriber msisdn=<SubscriberMSISDN>

# Second step: voice mail association
# (note, the format command is on one line).
STATE:CreateFullSubscriber,createVM
FORMAT_COMMAND:create voicemail msisdn=<VoiceMailMSISDN>
subscriber=<SubscriberMSISDN>
```

Now, launching the "CreateFullSubscriber" work order with NVSet filled with subscriber and voice mail MSISDNs will sequence the 2 interactions. If both commands return "OK", the work order will be successful.

See cartplugin package documentation for more information about this.

Using the TRN parser and complex guards

Some network element response will need more than an “OK” / “KO” response and will need more complex parsing / guards.

The TRN parser can be used to parse the output of a response and fill the work order NVSet with data extracted from the response from the network element.

Let's consider the following case on our above example; the subscriber creation can lead to the following responses:

The subscriber has been created successfully:
```
OK
```

The communication failed for some reason:
```
KO:[ some reason ]
```

The subscriber already exists:
```
KO:Already exist
```

If the subscriber already exist, it is not an error and we want the voice mail creation to be performed.

Inside the TRN parser, we can extract the result and error string and fill an NVSet parameter to indicate the decisions to take.

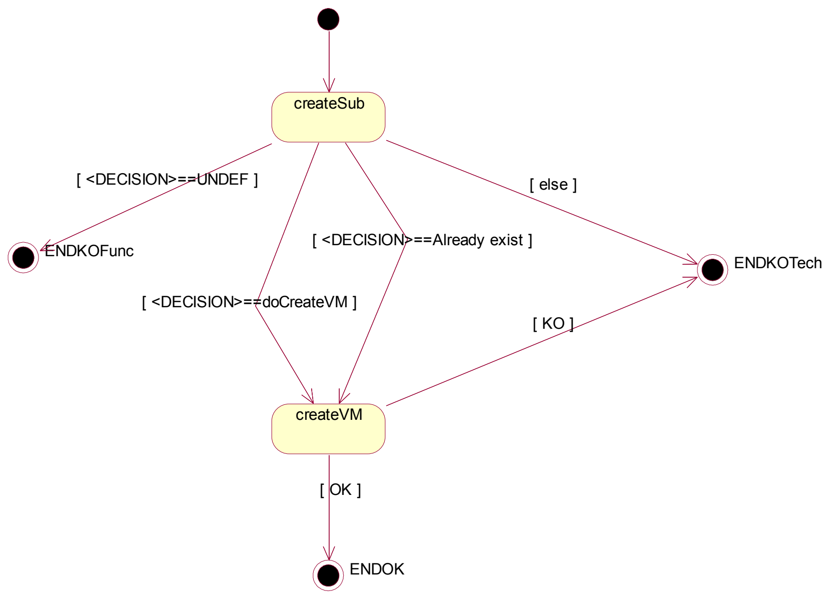
The TRN will therefore look like:

```
#
# Subscriber with voice mail creation work order:
#

# First step: subscriber creation:
STATE:CreateFullSubscriber,createSub
FORMAT_COMMAND:create subscriber msisdn=<SubscriberMSISDN>

# First step, parsing the response:
STATE:CreateFullSubscriber,createSub,Post
# We will use this attribute to make the decision in the XML
# state diagram:
FORMAT:<DECISION>=UNDEF

# If the response begins by “OK”, we succeeded in the creation
IF:<#NE_RESPONSE>=~^OK
         FORMAT:<DECISION>=doCreateVM
# Extract the error code, and set it inside the decision:
ELIF:<#NE_RESPONSE>=~KO:(..*)
         FORMAT:<DECISION>=<#REMATCH[1]>


# Second step: voice mail association
# (note, the format command is on one line).
STATE:CreateFullSubscriber,createVM
FORMAT_COMMAND:create voicemail msisdn=<VoiceMailMSISDN>
subscriber=<SubscriberMSISDN>
```

At the end of the parsing of the “createSub” state, the “DECISION” NVSet parameter can contain:

- “UNDEF” which should lead to a functional error: something went wrong with the network element and the TRN parser could not parse the output.
- “doCreateVM” which should go on to the voice mail creation.
- Or an equipment error code:
  - “Already exist” which should go on to the voice mail creation (as we specified that this is not an error).
  - Any other error code which should trigger a technical error (as this can be a communication failure, for example).

Therefore, the XML state diagram should look like:



Configuring the Cartridge

This chapter describes the creation and configuration parameters for CSL-enabled cartridge. This chapter will contain the following sections describing the overall sequence for setting up the cartridge:
- Note on the serial protocol.
- Create one or more types using the factory.
- Configure the types.
- Create one or more instances using each type.
- Configure the instances.

Note on the serial protocol

To be able to configure a cartridge using a CSL-enabled session layer using a particular serial protocol, users need to get the name of this session layer.
The name will be given in the serial protocol user documentation or inside the package's idlosdoc.

Moreover, some configuration parameters (mainly on instance configuration) will be added by the specific serial protocol.
For example the telnet serial protocol adds to the default CSL configuration its own parameters: “Port”, “Hostname” and “Timeout” parameters are CSL_SProTelnet only.

Create one or more types using the factory

We assume in this example that you will use the “CSL_SProTelnet” session layer, and that you will use the common formatter to format and parse the output of the telnet interface.

Parameter	Description	Default value
SessionLayer	The name of the SessionLayer for telnet CSL is “**CSL_SProTelnet**”.	None (mandatory)
DataProcessorBefore	Using the TRN formatter named CommonFormater.	None (mandatory)
DataProcessorAfter	Using the TRN parser named CommonParser	None (mandatory)

The following extract of TST file will create a cartridge type named “MyType”:

```
TRGT:PoolFactory
ACT:createType
```

Configuring the Cartridge : Create one or more types using the factory

```
NAME:MyType
NVSET:SessionLayer=CSL_SProTelnet|
DataProcessorBefore=CommonFormatter|
DataProcessorAfter=CommonParser
```

Once you have created a type, you can configure it and then use it to create cartridge instances.

Configure the type

The configure operation for a CSL cartridge type requires the default parameters:

Parameter	Description	Default value
XMLFilePath	Path of the XML file containing the work order flows (optional, but very likely to be there for CSL session layers).	“” (optional)
WOParamDescriptionFile	Path of the Work Order configuration file.	“” (optional)
TrnFilePath	Path to the TRN file containing the formatting and parsing directives for MML commands.	“” (optional)

Remarks:

- The configure paths can be absolute or relative.
- Both CommonFormatter and CommonParser share the same TRN file. See cmf package idlosdoc for more explanations.

The following command configures the cartridge type created in the previous section:

```
TRGT:PoolFactory:MyType
ACT:configure
NVSET:TrnFilePath=./conf/MyType.trn|
XMLFilePath=./conf/MyTypeWO.xml|
WOParamDescriptionFile=./conf/MyTypeWO.cfg
```

Creating the instance

The following command creates an instance of CSL-enabled cartridge named MyInstance:

```
TRGT:PoolFactory:MyType
ACT:createChild
NAME:MyInstance
NVSET:
```

Once you have created a cartridge instance, you must configure it before it can be enabled and unlocked for use.

Configuring the instance

This chapter will present:
- The default mandatory parameters coming from cartridge framework – see cartbase, cartplugin and cartpool documentation for more information.
- The *serial protocol component* configuration – see the specific documentation about it for more information.
- The parameters added by CSL – we will dedicate the next chapter to discover them in details.
- Example of configuration.

Default mandatory parameters from cartridge

In this type, we only have one mandatory parameter, added by cartpool:

Parameter	Description	Default value
NumberOfSessions	Number of sessions in the pool	None (mandatory)

Serial protocol component configuration

As an example, we will take SProTelnet parameters

Parameter	Description	Default value
Timeout	Timeout parameter of the recv operation on telnet interface.	None (mandatory)
Hostname	Hostname where the telnet server is located.	None (mandatory)
Port	Port to connect to on the remote host.	None (mandatory)

Note that by changing the serial protocol, taking pad for example, these parameters will not exist anymore and be replaced by specific pad parameters.
Cartridge and CSL parameters will still be needed in the configuration.

Parameters added by CSL

Here is a brief presentation of all parameters needed by the Common Session Layer:

Parameter	Description	Default value
LoginSequence	Description of the steps to log in the remote network element.	“” (optional)
LogoutSequence	Description of the steps to log out of the remote network element.	“” (optional)
LoginSleepAfterSend	Number of milliseconds to sleep after a send during login sequence.	0 (optional)
EarlyTokens	List of tokens indicating end of transmission in response buffer.	“” (optional)
MaxNumberOfAttempts	Number of retries for basic send / receive elements.	50 (optional)
PostCommandString	String to append to each buffer to send.	“” (optional)
PostCommandSkipString	String indicating that the PostCommandString should not be appended for this particular buffer to send.	“” (optional)

Parameter	Description	Default value
CloseSessionOnMaxAttempts	Boolean to indicate that a session should be disconnected if the maximum number of attempts is reached during a command execution. Either “true” or “false”.	“false” (optional)

Configuration example

Here is an example of configuration. This example, applied to “CSL_SPRoTelnet”, demonstrates how to connect a telnet cartridge to a unix telnet server:

```
TRGT:PoolFactory:MyType:MyInstance
ACT:configure
NVSET:NumberOfSession=1|LoginSequence=RECV,login:,SEND,[put your
login here],RECV,Password:,SEND,[put your password here],RECV,
[put your prompt here]|EarlyTokens=[put your prompt here]|
LogoutSequence=SEND,exit|PostCommandString=\n|Hostname=[put the
hostname of the remote server here]|Port=23|Timeout=5000
```

The example will be detailed in next chapter.

Feature and parameter description

This chapter will describe features of the Common Session Layers. In particular, the following aspects will be covered:

- Login and logout sequences.
- Closing on max attempts.
- Configuring exchanges.
- Handling early token absence.

Login and logout sequence

This chapter will present features and configuration of login and logout sequence.

Presentation

When a CSL-enabled session layer session is started, it first connects to the equipment, establishing the physical communication path.
For a telnet session, this means connecting to the server at the TCP level.

Then, based on configuration, it can execute a *login* sequence.
For a telnet session, this means responding to “Login:” / “Password:” prompts.

The session will be usable to send commands / receive responses only once this *login* sequence is done.

CSL sequence description

The command session layer presents sequences such as *login* or *logout* sequences in the same way. A CSL sequence is a list of { *operation*, *data* } couples where:

- The *operation* field is one of:
  - “SEND” to send data.
  - “RECV” to expect data.
- The *data* field is a fixed string.

For example, here are the steps to start a telnet exchange with a telnet server:

1. The client waits for the “Login:” prompt.
2. The client, having received the “Login:” prompt, sends its login (“me”).
3. The client waits for the “Password:” prompt.
4. The client sends its password (“p4ssw0rd”).

5. The client waits for the session prompt. “OK>” for example.

This will be translated into the following CSL sequence:

```
RECV,Login:,SEND,me,RECV,Password:,SEND,p4ssw0rd,RECV,OK>
```

Note that comas ',' separates both *operation* and *data* and also separates couples of { *operation*, *data* }.

Login sequence

The *login* sequence of a Common Session Layer will be executed after the connection with the equipment has been made.
Only once the login sequence is over and successful will the session be declared as enabled.
The login sequence can fail in the following cases:
1. The connection to the server is lost.
2. Data received from the server do not match “RECV” field.
3. Send or receive from the server fails in timeout or ko.

Note that the server will probably always accept data “SEND”. Only the response will not match the “RECV” field.

If we take back the telnet example, by giving a wrong password, the user prompt will never be received: the login sequence will fail on the “RECV,OK>” step.

Failure inside a login sequence will trigger the session disconnection: the session will be set back to “disabled” state.

To set the previously defined *login* sequence in the CSL configuration, put it in the “LoginSequence” NVSet parameter:

```
TRGT:PoolFactory:MyType:MyInstance
ACT:configure
NVSET:NumberOfSession=1|LoginSequence=RECV,login:,SEND,[put your
login here],RECV,Password:,SEND,[put your password here],RECV,
[put your prompt here]|EarlyTokens=[put your prompt here]|
LogoutSequence=SEND,exit|PostCommandString=\n|Hostname=[put the
hostname of the remote server here]|Port=23|Timeout=5000
```

Sleeping during login sequence

Some network elements cannot support quick data exchange during login sequence. They need time for processing received data.
There is a parameter allowing to introduce delays after a “SEND” is made: the “LoginSleepAfterSend” parameter. It accepts a delay in milliseconds.

You can add it to the configuration (however, it is useless for most network elements):

```
TRGT:PoolFactory:MyType:MyInstance
ACT:configure
NVSET:NumberOfSession=1|LoginSequence=RECV,login:,SEND,[put your
login here],RECV,Password:,SEND,[put your password here],RECV,
[put your prompt here]|EarlyTokens=[put your prompt here]|
LogoutSequence=SEND,exit|PostCommandString=\n|Hostname=[put the
hostname of the remote server here]|Port=23|Timeout=5000|
LoginSleepAfterSend=300
```

Logout sequence

The *logout* sequence of a Common Session Layer is an exchange that will be executed just before the connection with equipment is closed.

It will be triggered when the disable() operation on the session layer session is called, i.e. when the instance is disabling.

The logout sequence usually provides a way to shut down the interface nicely, by calling exit on the equipment for example. Note that this is a “best effort” feature: the connection to the equipment will be closed regardless of the status of this sequence.

Fill the “LogoutSequence” NVSet parameter to set a *logout* sequence:

```
TRGT:PoolFactory:MyType:MyInstance
ACT:configure
NVSET:NumberOfSession=1|LoginSequence=RECV,login:,SEND,[put your
login here],RECV,Password:,SEND,[put your password here],RECV,
[put your prompt here]|EarlyTokens=[put your prompt here]|
LogoutSequence=SEND,exit|PostCommandString=\n|Hostname=[put the
hostname of the remote server here]|Port=23|Timeout=5000
```

Note on more complex sequences

Users can specify more complex exchanges for login and logout sequences using *login* and *logout work orders*. See cartplugin for more information.

Closing session on max attempts

Description

CSL-enabled session layers can be configured to automatically close sessions if the max attempt number has been reached.
This means, for example, that all work orders ending in technical errors because a response never came will trigger a session close. You can configure automated reconnections using basic cartridge configuration parameters “RelogAttemptNumber” and “RelogTimeBackoff”.

Example

Here is an example of automatic closing and reconnection:

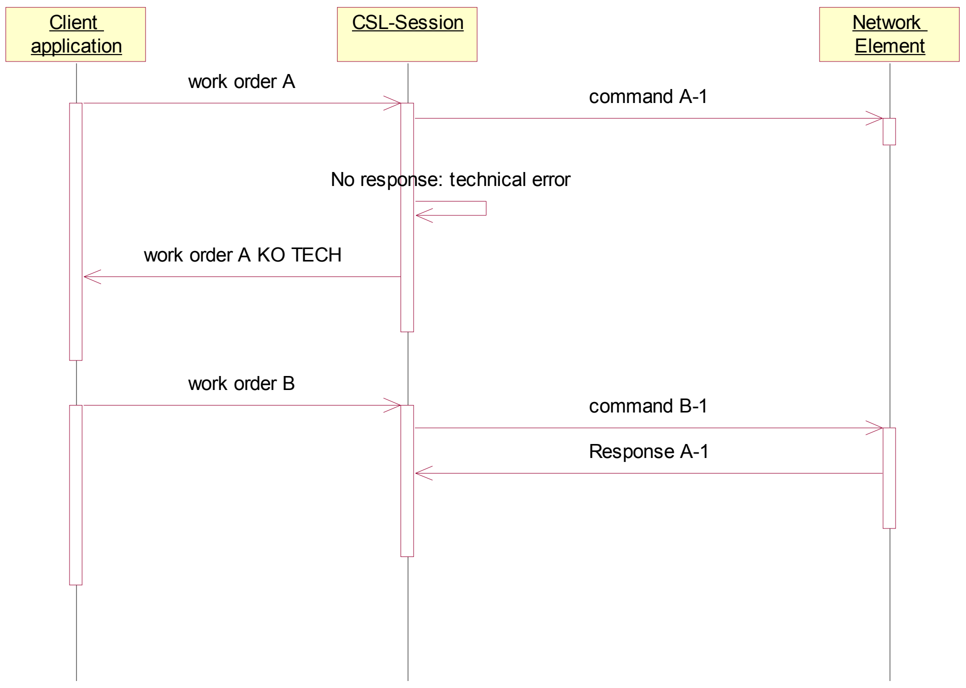
```
TRGT:PoolFactory:MyType:MyInstance
ACT:configure
NVSET:NumberOfSession=1|LoginSequence=RECV,login:,SEND,[put your
login here],RECV,Password:,SEND,[put your password here],RECV,
[put your prompt here]|EarlyTokens=[put your prompt here]|
LogoutSequence=SEND,exit|PostCommandString=\n|Hostname=[put the
hostname of the remote server here]|Port=23|Timeout=5000|
CloseSessionOnMaxAttempts=true|RelogAttemptNumber=5000|
RelogTimeBackoff=20
```

The previous configuration will:

- Close the session if max attempts is reached.
- Automatically attempt to relog 20 seconds after disconnection.
- Each session will try to relog up to 5000 times.

Use case

Using this feature avoids "late" response problems:



In the above example, work order B receives (and probably processes) response A-1 in place of some response for B-1. Closing the session on max attempts will disconnect the session after work order A KO TECH and make sure that response A-1 will not be received.

Configuring exchanges

Many parameters have an impact on the data exchange.

Append a string to sent data

A common feature on ASCII based protocols is to append a character (mainly '\n' for carriage return) to all sent data.
This can be done using “PostCommandString” parameter:

```
TRGT:PoolFactory:MyType:MyInstance
ACT:configure
NVSET:NumberOfSession=1|LoginSequence=RECV,login:,SEND,[put your
login here],RECV,Password:,SEND,[put your password here],RECV,
[put your prompt here]|EarlyTokens=[put your prompt here]|
LogoutSequence=SEND,exit|PostCommandString=\n|Hostname=[put the
hostname of the remote server here]|Port=23|Timeout=5000
```

This indicates that all commands to send will be appended with '\n' character. This also includes all sequences (login, passwords, etc).
For example:

```
RECV,Login:,SEND,me,RECV,Password:,SEND,p4ssw0rd,RECV,OK>
```

This will send “me\n” and “p4ssw0rd\n” to the server.
In the same vein of though, if the TRN contains a line:

```
FORMAT_COMMAND:ls <directoryPath>
```

The command sent to the network element can be “ls dir1/dir2\n” (based on the value of “directoryPath” when the TRN line is evaluated).
This parameter does not mean anything for received data.

Skipping the post command string

Note that the appending of the “PostCommandString” can be skipped by defining a “PostCommandSkipString”:

```
TRGT:PoolFactory:MyType:MyInstance
ACT:configure
NVSET:NumberOfSession=1|LoginSequence=RECV,login:,SEND,[put your
login here],RECV,Password:,SEND,[put your password here],RECV,
[put your prompt here]|EarlyTokens=[put your prompt here]|
PostCommandString=\n|PostCommandSkipString=SKIP_POST|
LogoutSequence=SEND,exitSKIP_POST|Hostname=[put the hostname of
the remote server here]|Port=23|Timeout=5000
```

With this configuration, all buffers to send ending with “SKIP_POST” string will not get the '\n' appended. This is also true for all sequences: in the example, the logout sequence will send the string “exit”, and not “exit\n”.

Configuring early tokens

The “EarlyTokens” parameter is one of the most important parameter in the configuration of a CSL-enabled session layer.

The “EarlyTokens” parameter contains a list of strings indicating that the receiving step is over for this command: it is a notification that the received buffer is complete.

For a telnet interface, it can be the prompt.
For a proprietary interface, it can be indications of success or failure for the sent command.

```
TRGT:PoolFactory:MyType:MyInstance
ACT:configure
NVSET:NumberOfSession=1|LoginSequence=RECV,login:,SEND,[put your
login here],RECV,Password:,SEND,[put your password here],RECV,
[put your prompt here]|EarlyTokens=[put your prompt here]|
PostCommandString=\n|LogoutSequence=SEND,exit|Hostname=[put the
hostname of the remote server here]|Port=23|Timeout=5000
```

In case no early token is found in the response buffer (in particular if you have not set one), the command will end in technical error. (see chapter on early tokens and XML state diagrams)
Note that early tokens ***do not*** apply to login, logout or pre-command sequence.

Pre-command setup

Some network elements require a specific interaction before each command to send. This is defined in the “PreCommandSequence” parameter. The content of the parameter is a sequence as in the *login* sequence:

```
TRGT:PoolFactory:MyType:MyInstance
ACT:configure
NVSET:NumberOfSession=1|LoginSequence=RECV,login:,SEND,[put your
login here],RECV,Password:,SEND,[put your password here],RECV,
[put your prompt here]|EarlyTokens=[put your prompt here]|
PreCommandSequence=SEND,echo OK,RECV,OK|PostCommandString=\n|
LogoutSequence=SEND,exit|Hostname=[put the hostname of the remote
server here]|Port=23|Timeout=5000
```

Any failure occurring during this pre-command will trigger:
- Session close (and reconnection if configured to do so),
- Technical error of the current command.

Handling early token absence

Some network elements will not indicate that the transmission of the response is over. To handle such cases, cartridge allows the following mode:
- Configure no early tokens. The session layer commands will all end in network error.
- However, the execution of the work order will go on, especially the execution of DataProcessorAfter, response parsers and XML state diagram:
  - If these executions are OK, then the network error on the session layer is unnoticed.
  - If one of these executions fails, the execution of the work order fails in network error.

This mode may be very ineffective. If we take the telnet example:

Feature and parameter description : Handling early token absence

- Not having an early token defined means that the telnet client will receive up to MaxAttemptNumber for *each* exchange.
- Each receive may block for up to “Timeout” milliseconds.

Depending on the configuration, this can lead to a huge delay.

Implementing and using the cartridge

This chapter contains the following sections which correspond to the needed steps to implement a CSL-enabled cartridge:

- Defining the work orders within XML state diagrams. (optional)
- Defining parameters given by cartridge configuration and parameters that will be given by client application.
- Writing a work order configuration file.
- Edit a TRN file where commands are written for each state in the work order.

Defining work order flows

A work order flow defines the steps that are required to perform a high- level command. You model a work order flow as an activity diagram in Rose. You must export the work order flows to an XML file, using the UNISYS Rose plugin. All of the work order flows for a specific NE cartridge must be contained in the same XML file.

When you model a work order flow in Rose, you must model the activity diagram within the context of a class with the stereotype `kabProcess`, which you define within a package with stereotype `kabPackage`.

A work order flow has:

- States
- Transitions

States

A work order flow has one Initial state and one or more Final states as well as any number of normal states. A normal state can represent a step in the current flow or reference another work order flow, enabling nested flows.

You can give a normal state any name you like; however, each normal state must have a name that is unique within its flow.

To indicate success in a Final state, label it ENDOK; to indicate failure, label it ENDKO. To represent a Network Error, label it ENDKOTech and to represent a Functionnal Error, label it ENDKOFunc. Note that ENDKO and ENDKOFunc are identical.

You can add state action plugin to your XML command tree in order to execute code before and after the session layer’s interaction.

For each step of the command state diagram, the TRN corresponding command will be invoked on the equipment.

Transitions

Transitions join the states, enabling the flow from one state to the next. To control the flow, use guarded transitions. As guard expressions you can use quoted strings or the reserved word **else**.It's up to the developer to put in the cartplugin::DEFAULT_RESPONSE the value that corresponds to the values of guards.You must model the flow such that for any given input, there is only one possible route through the flow.

You must model all of the work order flows for one NE cartridge in the same .MDL file so that you can export them all to the same .XML file.

To export a .XML file, use the command Tools > Export UML and select the XML 1.1 option.

For more details on how to create and use work order flows, please refer to the cartplugin API Documentattion and to the CART241SOL README file.

CSL-enabled session layer input parameters

CSL-enabled session layers have only one input parameter:

Name Type Description	Enumeration values	Required Cardinality	Remark
#NE_COMMAND String Buffer to send to the network element.	None	Mandatory Single value	None

CSL-enabled session layers do not have supported commands: they will not take into account the commandName parameter of processCommand() method.
Regardless of the commandName, CSL will send the buffer contained in "#NE_COMMAND" and will return in "#NE_RESPONSE" the buffer sent back by the network element.

Defining configurable parameters

CSL-enabled session layers will need the "#NE_COMMAND" parameter to process a command on a network element. This parameter is likely to be formatted in a TRN file, based on other parameters:

```
FORMAT_COMMAND:ls <directoryPath>
```

In this example, the state where this formatting directive is given will require the “directoryPath” parameter to be filled in the NVSet. The “directoryPath” can be defined in one of the following ways, depending on the application needs:

- The client application can set some of them inside the Work Order NVSet.
- The parameters can be defined as “additional” parameters on the cartridge type configuration: they will then appear as parameters for the cartridge instance configuration, and all Work Orders NVSet will be modified to contain these parameters.

Creating the work order configuration file

The work order description file (.cfg) will be loaded during Type's configure() operation, under parameter " WOParamDescriptionFile ". In this file, you define the work order names and for each work order you specify the parameters given by the client application.

N.B: **For maintenance purposes, it is very important that you define all parameters that the client application should provide. This will allow early detection of errors.**

Here is an example of parameter settings, assuming that a client application provides some of them:

```
WOPARAM:directoryPath|DT_String|C_None|Card_Single|
Req_Mandatory|||Directory path for the command|0|0
```

Writing the TRN file

Each substep in a TRN file is associated with a step in a work order flow.

The TRN file will be written using a specified format, and then it will be loaded during Type's configure() operation, under parameter "TrnFilePath".

Each type of network element will probably require a custom made TRN file.

Troubleshooting / FAQ

This chapter will deal with frequent problems and frequently asked questions. Particularly, this chapter will address:

- How to debug login sequences ?
- How to troubleshot slow exchanges ?
- When and how does CSL publish logs ?
- How to configure a login or logout sequence with a special character inside ?
- How to develop *serial protocol components* ?
- Does CSL handles timeout ?

How to debug login sequences ?

Details of the execution of login and logout sequences can be found in cartridge order tracing (cartOrderTracing.*.log) under “NA” work order name.

SEND and RECV fields of login and logout exchanges must be configured to match exactly what has to be sent and received from the equipment. If they do not match (especially RECV fields), the connection will be dropped and the enable operation will be considered as failed.

How to troubleshot slow exchanges ?

If you experience slow exchange or if nothing appears in cartOrderTracing.*.log, you might have a misconfiguration:

- The “MaxAttemptNumber” parameter might be very high,
- And/or some timeout parameter on the *serial protocol component* specific parameters might be too high.

We recommend lowering both parameters to something more acceptable in terms of response time.

When and how does CSL publish logs ?

Publication time

CSL-enabled session layers will publish logs:
- When the work order is finished in case of a work order execution.
- When the login or logout sequence is done in case of an enable() or disable() operation.

In any case, logs are published at the end of a sequence of operations (sends and receives, inside a work order or not). Therefore, depending on configuration parameters of the instance, the publication time can vary. See How to troubleshot slow exchanges ? chapter.

Published logs

CSL-enabled session layers will publish the following logs:
- Session connection / disconnection notification.
- Session pre-command failures.
- Session forced closing on max attempts.
- Session max attempts reached.
- Buffer sent and received to / from network element.
- Errors / timeout on send / recv operations.
- Early token detection.

Sent/received buffer format

When tracing sent or received buffer, non printable or special characters are converted on the fly:
- Carriage return, Line feed or tabulation are printed as "\n", "\r" and "\t" (i.e. '\' char *plus* letter).
- All non printable chars (returned as such by isprint() operating system C method), along with '|' (pipe), ',' (coma) are converted to '\xxx' where xxx is the octal code for the character from the ASCII table.

Note that the value of the NVSet parameters "#NE_COMMAND" or "#NE_RESPONSE" is not modified, only traces are modified.

How to configure a login or logout sequence with a special character inside ?

To send, for example a coma ',' as part as a login, one can use:

```
RECV,Login:,SEND,my\054login,RECV,Password:,SEND,p4ssw0rd,RECV,OK
>
```

This will make the login sent as “my,login”. 054 is the ASCII code of ',' in octal (see ASCII man pages on your system for char tables).
Note that:

- All chars can be specified using this notation (printable or not printable).
- This also applies to TRN files and XML state machine guards.

How to develop serial protocol components ?

To develop *serial protocol components* see csl package idlos documentation.

Does CSL handles timeout ?

The Common Session Layer does not handle operation timeout by itself, in the sense that there is no timeout mechanism built in CSL.

CSL lets *serial protocol component* methods (send or recv) handle timeout. If the protocol used as the serial protocol supports timeout, then the *serial protocol component* will expose this parameter and let you configure it.

For example in the SProTelnet component:

- “Timeout” appears as a configuration parameter.
- This parameter is used for the TCP recv() low level operation.

Therefore, the CSL-enalbed “CSL_SProTelnet” session layers handles timeout. But another CSL-enabled session layer might not handle it.

External resources

- Not applicable

References

- cartpool API idlos documentation.
- cartplugin API idlos documentation.
- cmf API idlos documentation.

Dependencies

- OS424
- CART24
- FTP225